Paper 3722-2019

Use the Advantage of INDEXes Even If a WHERE Clause Contains an OR Condition

Bartosz Jabłoński
Warsaw University of Technology / Citibank Europe PLC Poland

ABSTRACT

The advantage of an indexed Base SAS® engine's dataset is that when the dataset is queried with a `WHERE` clause the process may be optimized and index-subsetting will work faster than a sequential read. Unfortunately the Base SAS engine is unable to use more than one index at a time to optimize a `WHERE` clause. Especially if a `WHERE` clause contains an `OR` condition between two different (indexed) variables the Base SAS engine won't optimize it and will execute a sequential read.

The main idea of this paper is to offer a solution to the following question: How to handle a situation in which we have a `WHERE` clause with an `OR` condition between two different indexed variables and want to use the advantage of indexing to subset the data faster? The solution is datastep based and uses hash-tables. But foremost it is both simple to implement and efficient.

INTRODUCTION

When we are working with datasets in the SAS Base engine we often try to use some optimization techniques which allow us to improve performance. There are dozens of such techniques but some of the most basic approaches could be described with the following steps:

- narrow the data, i.e. select an observation only `IF` it is necessary for the process,
- even better, use a `WHERE` clause instead of a subsetting `IF` statement, as you will reduce the data before it is inserted into PDV,
- if a `WHERE` clause subsets only a small part of the dataset (and the dataset is big enough, i.e. spans across more than 3 pages) add an `INDEX` to the dataset,
- if your `INDEX` file is big and the dataset is quite "static" (i.e. doesn't change to often) consider Mark Keintz's compressed indexes (see [**8**]).

Unfortunately there's a limit to index usage as SAS Base engine can't use more than one index to optimize a `WHERE` clause. It is explained in the question 22: "Why can't an index be used if there is an `OR` in the `WHERE` expression?" in Billy Clifford's paper [**6**].

But are we 100% sure that we can't do this, i.e. to use the advantage of `INDEX`es when a `WHERE` clause contains an `OR` condition? In fact we can. If we incorporate some small additional programming effort we are able to overcome the obstacle described above.

There are two basic steps behind the process. The first is to realize that a `WHERE` clause with an `OR` condition can be split into separate clauses which can be executed independently (and use different `INDEX`es). The second is to realize that if we can efficiently manage information on which observations have already been read we won't have a problem with consolidating the data and will avoid potential duplicates.

For the sake of clarity, from now on, any further phrases such as "SAS is doing something" will be related to the Base SAS engine unless explicitly noted.

TOOLS

Before we start the main topic let's take a quick look at two basic concepts we are going to work with, i.e. indexes and hash-tables.

Indexes. The concept of SAS index, from a user point of view, is very simple and intuitive. We can think of an index as a list of "key-value" pairs. "Keys" are values of the variable on which the index is built. "Values" are lists of row identifiers, a.k.a. RIDs, which are the pointers to location of observations containing a given value in a dataset. The most natural analogy would be a book and... its index. For example, if our dataset looks as follows:

A dataset with two variables

Obs.	VarA	VarB
1	A	10
2	B	20
3	C	10
4	A	20
5	B	10
6	C	20
7	A	10

we can think of an index constructed for variable VarA as:

An index for variable `VarA`

Key:	{RIDs}
A:	{1, 4, 7}
B:	{2, 5}
C:	{3, 6}

and for variable VarB as:

An index for variable `VarB`

Key:	{RIDs}
10:	{1, 3, 5, 7}
20:	{2, 4, 6}

We have already mentioned that indexes are used to optimize data selection in `WHERE` clauses. When we write the code:

```
WHERE VarA = "B";
```

SAS estimates the number of observations read by the clause from the dataset, and if it is worth it, SAS uses the index. What does it mean "uses the index"? Well, instead of a sequential read through the dataset, SAS will look at a list of record identifiers for that given value `"B"` of variable `VarA` and will read only the observations pointed by RIDs.

Just as a reminder, this is only "a user point of view". Of course, under the hood it is more complex than the description above. Starting for example with the fact that indexes are stored in a separate file, and they are tree-shaped data structures, and the "estimation of the number of observations to be read" is a complicated process, and eventually optimization of `WHERE` clauses is not the only purpose of indexes existence. But the intuition we already have is good enough for the beginning. Very good references to discover indexes in details are: Billy Clifford's paper [**6**] and Michael A. Raithel's book [**7**], and of course SAS on-line documentation.

Hash-tables. The concept of a hash-table is very user friendly as long as we start with good intuition. Users which are not familiar with object oriented programming notation may, at the first glance, consider hash-table's syntax a bit awkward, but do not judge a book by its cover!

From a user perspective a hash-table can be considered as a younger and smarter sibling of a classical, well known, SAS temporary array. Let's take a quick look at arrays and declare a temporary array `ARR`. We can do it for example by calling the following code:

```
array ARR[6] $ 1 _temporary_;
```

We can visualise `ARR` as a pre-allocated, in-memory, and fixed-size set of adjacent cells, with integer pointer addressing each cell, waiting for the data. Like in the figure below:

A temporary array `ARR` waiting for the data

cell key	cell value
[1]	" "
[2]	" "
[3]	" "
[4]	" "
[5]	" "
[6]	" "

To populate `ARR` with the data we can run the following code:

```
k = 2; v = "B"; ARR[k] = v;    ❶
 k = 1; v = "A"; ARR[k] = v;   ❷
  k = 4; v = "D"; ARR[k] = v;  ❸
```

and after each line of code the array can be visualised as presented in the figure below:

temporary array ARR populated with the data

after ❶ cell key	after ❶ cell value	after ❷ cell key	after ❷ cell value	after ❸ cell key	after ❸ cell value
[1]	" "	[1]	"A"	[1]	"A"
[2]	"B"	[2]	"B"	[2]	"B"
[3]	" "	[3]	" "	[3]	" "
[4]	" "	[4]	" "	[4]	"D"
[5]	" "	[5]	" "	[5]	" "
[6]	" "	[6]	" "	[6]	" "

To retrieve a cell's value we are using a corresponding key's value in array's reference as for example:

```
k = 2; v = ARR[k]; put v=;
```

and as a result in the SAS log we will see: `v=B` printed out.

In case of a hash-table (again, from a user point of view) the process, modulo the syntax, looks similar. Let's declare a hash-table `HSH`. We can do it for example by calling the following code:

```
length k 8 v $ 1;
declare hash HSH(ordered:"ascending",
                 hashexp:8);
HSH.DefineKey("k");
HSH.DefineData("v");
HSH.DefineDone();
```

Before we continue a note about syntax's analogies. In the `declare hash` statement we are giving a hash-table a name (it is an `array` statement's analogy). With the `ordered` option we are forcing keys to be in the ascending order (in the array keys are ordered by default since they are ascending integers). The `hashexp` option establishes maximum size of a hash-table (it could be *very very* loosely compared to array's size declaration). The `.DefineKey()` method indicates variables which are used as a key (an analogy of `k` in the `v = ARR[k]` code) and the `.DefineData()` method indicates variables which are used to hold the data portion (an analogy of `v` in the `v = ARR[k]`). And the `.DefineDone()` method is the equivalent of a semicolon at the end of array's definition.

We can visualise `HSH` as a *not*-pre-allocated, in-memory, and *not*-fixed-size set of "key-data" pairs, with (not necessary integer) key-pointers addressing each data portion. Hence after the declaration `HSH` looks like in the figure below and it is awaiting to be populated with the data.

Hash-table `HSH`
waiting for the data

key	data
*	*

The process of inserting data into `HSH` uses the `.add()` method[1] and to populate `HSH` with the data we can run the following code:

```
k = 2; v = "B"; HSH.add();     ❶
 k = 1; v = "A"; HSH.add();    ❷
  k = 4; v = "D"; HSH.add();   ❸
```

and after each line of code the hash-table can be visualised as presented in the figure below (notice how the order of keys is changing due to the `ordered:"ascending"` tag):

[1] A method in object oriented programming terminology may be considered as a function associated with an object.

hash-table `HSH`
populated with the data

after ❶

key	data
[2]	"B"

after ❷

key	data
[1]	"A"
[2]	"B"

after ❸

key	data
[1]	"A"
[2]	"B"
[4]	"D"

When a hash-table has been populated retrieving data is as simple as $2+2$. All we need to do is to set the key's value and call `HSH`'s `.find()` method, as in the following example:

```
k = 2; HSH.find(); put v=;
```

and in the SAS log we will see: `v=B` printed out. We don't even have to use any assignment statement since the hash-table will handle it itself. If the `.find()` method will be successful then variable `v` will be populated with data automatically.

As a side note, the above process of adding the data to the hash-table `HSH` can also be executed with a do loop in a similar fashion as with arrays. Code for arrays would look as follows:

```
k = 0;
do v = "A", "B", "C", "D", "E", "F";
  k + 1;
  ARR[k] = v;
end;
```

while for hash-tables it would be:

```
k = 0;
do v = "A", "B", "C", "D", "E", "F";
  k + 1;
  HSH.add();
end;
```

A hash-table doesn't have to use integers as keys and, what is even more comfortable, we can have a more complex data portion than a single cell. For example we can declare a hash-table `HT` and populate it with data in the following way:

Code:

```
length k1 8 k2 $ 1 v1 $ 1 v2 8;
declare hash HT(ordered:"ascending");
HT.DefineKey("k1", "k2");
HT.DefineData("v1", "v2", "k1");
HT.DefineDone();

k1 = 1; k2 = "m"; v1 = "A"; v2 = 13; HT.add();
k1 = 2; k2 = "i"; v1 = "B"; v2 = 17; HT.add();
k1 = 3; k2 = "n"; v1 = "C"; v2 = 42; HT.add();
k1 = 4; k2 = "i"; v1 = "D"; v2 = 66; HT.add();
```

```
k1 = 5; k2 = "p"; v1 = "E"; v2 = 78; HT.add();
k1 = 6; k2 = "w"; v1 = "F"; v2 = 82; HT.add();
```

Visualisation:

hash-table HT

key k1, k2	data v1, v2, k1
[1, "m"]	"A", 13, 1
[2, " i "]	"B", 17, 2
[3, " n "]	"C", 42, 3
[4, " i "]	"D", 66, 4
[5, " p "]	"E", 78, 5
[6, "w"]	"F", 82, 6

What's even more useful is that hash-tables allow to load data straight from an external dataset within `declare hash` statement. For example, assuming that dataset `work.SomeDataset` contains tree numeric variables `k`, `d1`, and `d2` we could load it into a hash-table with following code:

```
length k d1 d2 8;
declare hash HfrmDS(dataset:"work.SomeDataset");
HfrmDS.DefineKey("k");
HfrmDS.DefineData("d1", "d2");
HfrmDS.DefineDone();
```

So, the punch line here is that a hash-table is a flexible and dynamically allocated data structure, with a very efficient data access time and a "dictionary" kind of behaviour.

Again, as a reminder, this is only "a user point of view". As in the case of indexes, also in the case of hash-tables there is much more happening under the hood than in the description above. Starting for example with the fact that there is an internal hashing function involved, and data are kept in tree-shaped data structures (AVL-trees). But also in this case the intuition we already have is good enough. References to discover hash-tables in more details are: Paul Dorfman's paper [**5**], Paul Dorfman's and Don Henderson's book [**4**], Chris Schacherer's paper [**3**], Paul Dorfman's and Koen Vyverman's paper [**2**], Art Carpenter's book [**1**], and of course SAS on-line documentation.

THE PROCESS

Now, when we covered all prerequisites, we can begin the process of modifying our code in a way that will allow us to handle a `WHERE` clause with an `OR` condition and at the same time use the advantage of `INDEX`es.

The Dataset. At the very beginning let's turn-on some additional options which will give us extended logging features.

```
options
  FULLSTIMER   ❶
  MSGLEVEL = I ❷
;
```

In ❶ we specify whether to write extended system performance statistics to the SAS log (e.g memory usage, real time, cpu time, etc.) In ❷ we specify the level of details in messages that are written to the SAS log and value "`I`" indicates to print additional notes pertaining to index usage, merge processing, sort utilities, etc.

Using *Wikipedia*[2] as a data base we are going to prepare a randomly ordered (❸) dataset of countries names (248 observations) which will be used as a base for data preparation in the main code.

```
libname mysets BASE "...";

data mysets.countries;
  infile cards dlm = '0A'x;
  input country $ :50.;
  call streaminit(2222);
  sort = rand("uniform"); ❸
cards;
Afghanistan [AFG]
Aland Islands [ALA]
Albania [ALB]
...
Virgin Islands, US [VIR]
Wallis and Futuna Islands  [WLF]
Western Sahara [ESH]
Yemen [YEM]
Zambia [ZMB]
Zimbabwe [ZWE]
;
run;

proc sort
  data = MySets.Countries
  out  = MySets.Countries(drop = sort)
;
  by sort;
run;
```

As the next step we are going to create a bigger dataset which we will use in the process.

```
data mysets.INDEXX_OR(
  INDEX = (
    country ❹
    date    ❺
```

[2] https://en.wikipedia.org/wiki/ISO_3166-1

```
274   )
275 );
276  set ❻
277   mysets.countries
...
288   mysets.countries
289  ;
290
291  format date yymmdds10.;
292  do date = '1jan1960'd to '28apr2019'd; ❼
293   y = year(date);
294   m = month(date);
295   d = day(date);
296
297   call streaminit(123); ❽
298   measurement = 456+round(rand("Normal")*78); ❾
299   output;
300
301   if rand("Uniform") > 0.9 then output; ❿
302  end;
303 run /*cancel*/ ;
```

The `INDEXX_OR` dataset has two simple indexes: `country` (❹) and `date` (❺), created in lines 271 to 274. The dataset is build in the following way: for each country (repeated a dozen times ❻) a bunch of records, with dates (❼) and random measurements (❾, the ❽ sets seed for `rand()` function) and with about 10% of "natural" duplicates (❿), is generated. Two following procedures will give us the dataset's shape and metadata.

```
304 proc contents
305   data = mysets.INDEXX_OR;
306 run;
307 proc print
308   data = mysets.INDEXX_OR(obs=3);
309   where country = 'Yemen [YEM]'
310     and date = '28apr2019'd
311   ;
312 run;
```

The output shows:

```
The CONTENTS Procedure

Data Set Name          MYSETS.INDEXX_OR
Member Type            DATA
Engine                 V9
Created                04/28/2019 09:00:00
Last Modified          04/28/2019 09:00:00
Data Representation    WINDOWS_64
Encoding               utf-8
Observations           70935765
Variables              6
Indexes                2
Observation Length     96
Deleted Observations   0
Compressed             NO
Sorted                 NO

Engine/Host Dependent Information
Data Set Page Size            65536
Number of Data Set Pages      104166
First Data Page               1
Max Obs per Page              681
Obs in First Data Page        665
Index File Page Size          4096
Number of Index File Pages    293461
Number of Data Set Repairs    0
ExtendObsCounter              YES
Filename                      indexx_or.sas7bdat
Release Created               9.0401M4
Host Created                  X64_10PRO
Owner Name                    sasmaniandevil
File Size                     6GB
File Size (bytes)             6826688512

Alphabetic List of Variables and Attributes
#     Variable       Type    Len    Format
1     country        Char     50
5     d              Num       8
2     date           Num       8    YYMMDDS10.
4     m              Num       8
6     measurement    Num       8
3     y              Num       8

Alphabetic List of Indexes and Attributes
#     Index       # of Unique Values
1     country        248
2     date         21668

The PRINT Procedure

     Obs      country        date     y  m  d measu-
                                                rement
 1763886 Yemen [YEM] 2019/04/28 2019 4 28     398
 7675202 Yemen [YEM] 2019/04/28 2019 4 28     328
13585810 Yemen [YEM] 2019/04/28 2019 4 28     388
```

Now let's do some testing.

Indexes usage test. We are going to summarize measurements in variable `SoM` (❶) and count them in variable `i` (❷) for observations selected with a `WHERE` clause. We will do it for both SQL and datastep processing. To prove that `proc SQL` really uses an index to work the `WHERE` clause out we will run the following code:

```
proc sql;
  select
    sum(measurement) as SoM format best32. ❶
  , count(1) as i                          ❷
  from
    mysets.INDEXX_OR
  where
    country = 'Poland [POL]'
  ;
quit;
```

When we look into the SAS log, thanks to the MSGLEVEL = I option, we can see the following information:

```
INFO: Index country selected for
      WHERE clause optimization
```

Just for completeness, if we use a WHERE clause to subset data in the datastep:

```
data _NULL_;
  set mysets.INDEXX_OR END = eof;
  where
    date between '01may2015'd and '30may2015'd
  ;

  SoM + measurement;
  i + 1;

  if eof then
    do;
      put SoM= best32. i=;
    end;
run;
```

we will receive an equivalent INFO notification relating to index date.

In the next test we will see that when the WHERE clause contains the OR condition on two different variables SAS won't use any index to optimize subsetting, regardless we use the WHERE clause in proc SQL (❸) or in a datastep (❹).

```
proc sql;
  select
    sum(measurement) as SoM format best32.
  , count(1) as i
  from
    mysets.INDEXX_OR
  where ❸
    date between '01may2015'd and '30may2015'd
    OR
    country = 'Poland [POL]'
  ;
quit;

data _NULL_;
  set mysets.INDEXX_OR END = eof;
  where ❹
    date between '01may2015'd and '30may2015'd
    OR
    country = 'Poland [POL]'
  ;

  SoM + measurement;
  i + 1;

  if eof then
  do;
    put SoM= best32. i=;
  end;
run;
```

After running the above code the SAS log contains the following notes for proc SQL:

```
NOTE: PROCEDURE SQL used (Total process time):
      real time           1:32.40
      user cpu time       8.79 seconds
      system cpu time     5.45 seconds
      memory              5478.53k
      OS Memory           20724.00k
```

and for the datastep:

```
SoM=174856439 i=383492
NOTE: There were 383492 observations read
      from the data set MYSETS.INDEXX_OR.
      WHERE (date>='01MAY2015'D
             and date<='30MAY2015'D)
             or
            (country='Poland [POL]');
NOTE: DATA statement used (Total process time):
      real time           48.16 seconds
      user cpu time       5.39 seconds
      system cpu time     4.29 seconds
      memory              724.56k
      OS Memory           15856.00k
```

There is no information about index usage during code execution. In both cases a sequential read took place.

General Overview. Ok, so how to solve the "OR issue"? The solution is datastep based and uses hash-tables, but we will go through it step-by-step starting with arrays approach and than jumping into "hashes".

Before we dive into details, first and the most important thing is to realise that we can split the WHERE clause around the OR condition into two separate clauses. After that we can execute both WHERE clauses independently - what ensures that indexes will be used since we are having only simple conditions. Eventually, as the final step, we have to somehow bring the results (i.e. subsetted data) together. But we have to do it in such a way that the combined dataset will not contain duplicated observations coming from both WHERE clauses. The idea to prevent duplicates is to keep the record of already read observations.

Example. Let's consider the following WHERE clause

```
WHERE VarL="B" OR VarN=20;
```

and the following dataset:

A dataset with two indexed variables

Obs.	VarL	VarN
1	A	10
2	A	10
3	A	20
4	B	20
5	B	20
6	B	30
7	C	30
8	C	30
9	C	40

WHERE VarL="B" or VarN=20

WHERE VarL="B"

WHERE VarN=20

with both variables indexed. If we split the WHERE clause into two independent clauses: WHERE VarL="B" and WHERE VarN=20, as in the figure above, and execute them under two separate set statements in one datastep we will get duplicated records in the produced dataset. In the figure below they are marked with ⊛ symbol. The "Current Obs." column keeps track of the observation's number that was read-in.

Dataset with observations read by
WHERE VarL="B" and WHERE VarN=20

Current Obs.	VarL	VarN	
4	B	20	✔
5	B	20	✔
6	B	30	✔
3	A	20	✔
4	B	20	⊛
5	B	20	⊛

As we wrote the idea is to keep a record of observations read during execution of the first part of the WHERE clause. Hence, in the example we are considering, after executing WHERE VarL="B" clause we have the following observations in the output dataset:

Observations read with WHERE VarL="B"

Current Obs.	VarL	VarN	
4	B	20	✔
5	B	20	✔
6	B	30	✔

and a list of observations numbers read from the input dataset:

A list of read observations

observation number	"4"	"5"	"6"

When we execute the second clause each time before sending an observation to the output dataset we are checking if its number is on the list. In the example we are using the WHERE VarN=20 clause which fetches observations 3, 4, and 5. Since observations number 4 and 5 were already on the list hence they are not outputted and only the observation number 3 is. The final dataset contains only four observations marked with tick-mark ☑ for observations from the first WHERE clause part and with ✔ for observations from the second and duplicated observations omitted (marked with ✖).

Observations added by WHERE VarN=20

Current Obs.	VarL	VarN	
4	B	20	☑
5	B	20	☑
6	B	30	☑
3	A	20	✔
~~4~~	~~B~~	~~20~~	✖
~~5~~	~~B~~	~~20~~	✖

Now when we have a general overview of the process and we tested it with an example we can jump right into the code.

Programming. The first attempt considers using a temporary ARRAY. To be able to do that we have to do some "preprocessing". We have to get the number of observations to set the ARRAY's size and in consequence allocate memory.

```
data _null_;
 if 0 then set mysets.INDEXX_OR nobs = nobs;
 call symputx("_NOBS_", nobs, "G");
 stop;
run;
```

Having the metadata (i.e. the number of observations, _NOBS_ macrovariable) collected we are:

❶ declaring a temporary ARRAY to be a list to mark visited observations,
❷ executing a DoW-loop[3] in which we are reading-in data for the first part of our WHERE clause (lines 374 to 384),
❸ using the CUROBS option to create a variable that contains the observation number that was just read from the dataset,
❹ marking the ARRAY's cell which key equals to the current value of curobs variable,
❺ starting to aggregate the data (lines 382 and 383),
❻ executing a second DoW-loop in which we are reading-in data for the second part of the WHERE clause (lines 387 to 399),
❼ verifying if a visited observation was already read and if that's the case, going to the next iteration and skipping the aggregation,
❽ and eventually if the visited observation wasn't already read marking it in the temporary array (line 395, it allows to add third, fourth and further WHERE conditions in a very simple way just by copying the second DoW-loop and changing the condition) and updating aggregated data (lines 396 and 397).

```
371 data _NULL_;
372  ARRAY _obs_[&_NOBS_.] _temporary_; ❶
373
374  do until(eof); ❷
375   set
376     mysets.INDEXX_OR END=eof CUROBS=curobs ❸
377   ;
378   where date between '01may2015'd
379                  and '30may2015'd;
380
381   _obs_[curobs] = 1; ❹
382   SoM + measurement; ❺
383   i + 1;
384  end;
385
386  eof = 0;
387  do until(eof); ❻
388   set
389     mysets.INDEXX_OR END=eof CUROBS=curobs
390   ;
391   where country = 'Poland [POL]';
392
393   if _obs_[curobs] NE 1 then ❼
394    do;
395     _obs_[curobs] = 1; ❽
396     SoM + measurement;
397     i + 1;
398    end;
399  end;
400
401  put SoM= best32. i=;
402  stop;
403 run;
```

The result is the same as in the case of the previous, index-less, ones but now the SAS log shows totally different notes and infos.

```
INFO: Index date selected for WHERE
      clause optimization.
INFO: Index country selected for WHERE
      clause optimization.

SoM=174856439 i=383492
NOTE: There were 98206 observations read
      from the data set MYSETS.INDEXX_OR.
      WHERE (date>='01MAY2015'D
             and
             date<='30MAY2015'D);
NOTE: There were 285681 observations read
      from the data set MYSETS.INDEXX_OR.
      WHERE country='Poland [POL]';
NOTE: DATA statement used (Total process time):
      real time           4.78 seconds
      user cpu time       0.68 seconds
      system cpu time     1.78 seconds
      memory              555102.21k
      OS Memory           570044.00k
```

We can see that indexes were used, which decreased the execution time. Unfortunately the ARRAY approach has one drawback. The consequence of using a temporary array is that we have to preallocate the memory to handle markers for all observations even-though only a small part of ARRAY's cells will be used, which is inefficient.

A solution for ARRAY's memory issue would be a data structure which can dynamically modify its size. And in such a case a hash-table appears to be the perfect candidate. A hash-table allows us to add elements without previous memory allocation and in terms of searching works very efficiently (not as fast as array's direct access but fast enough).

To use a hash-table our previous code needs only slight changes:

❶ an array declaration is replaced with a hash-table declaration (lines 406 to 409),

[3] See Paul Dorfman's paper [9] to learn more details about this wonderful programming technique.

- ❷ in the first DoW-loop direct marking of visited observations is replaced with hash-table's `.add()` method (line 416),
- ❸ in the second DoW-loop in the `if` statement direct access is replaced with hash-table's `.find()` method (line 426),
- ❹ in the second DoW-loop direct marking of visited observations is replaced with hash-table's `.add()` method (line 428, our previous observation, made in the `array` approach, about adding new conditions remains).

```
404 data _NULL_;
405
406  length curobs 8;
407  declare HASH _obs_(hashexp:16); ❶
408  _obs_.DefineKey("curobs");
409  _obs_.DefineDone();
410
411
412  do until(eof);
413   set mysets.INDEXX_OR END=eof CUROBS=curobs;
414   where date between '01may2015'd
415                  and '30may2015'd;
416   rc = _obs_.add(); ❷
417   SoM + measurement;
418   i + 1;
419  end;
420
421  eof = 0;
422  do until(eof);
423   set mysets.INDEXX_OR END=eof CUROBS=curobs;
424   where country = 'Poland [POL]';
425
426   if _obs_.find() NE 0 then ❸
427    do;
428     rc = _obs_.add(); ❹
429     SoM + measurement;
430     i + 1;
431    end;
432  end;
433
434  put SoM= best32. i=;
435  stop;
436 run;
```

The result is the same as in the previous cases. The log shows following notes and infos.

```
INFO: Index date selected for WHERE
      clause optimization.
INFO: Index country selected for WHERE
      clause optimization.

SoM=174856439 i=383492
```

```
NOTE: There were 98206 observations read
      from the data set MYSETS.INDEXX_OR.
      WHERE (date>='01MAY2015'D
             and
             date<='30MAY2015'D);
NOTE: There were 285681 observations read
      from the data set MYSETS.INDEXX_OR.
      WHERE country='Poland [POL]';
NOTE: DATA statement used (Total process time):
      real time           1.99 seconds
      user cpu time       0.68 seconds
      system cpu time     1.31 seconds
      memory              26409.81k
      OS Memory           41148.00k
```

Again we can see that indexes were used, which significantly decreased the execution time. In case of the hash-table approach memory footprint is much smaller than in the array case (~26MB vs. ~555MB). To be clear, the memory footprint is still bigger than the one from index-less processing but now it looks like fair trade-off between time and RAM.

There is another approach which uses hash-tables, simplifies the code and makes it execute faster. Just to differentiate it from the previous approach let's name it "hash approach 2". Changes in the code are:

- ❶ two DoW-loops are replaced with a single `set` statement with input dataset used twice,
- ❷ and ❸ the `WHERE` clauses are moved into dataset options (lines 450 and 454),
- ❹ the `.find()` method is replaced with the `.check()` method which doesn't retrieve the data but only checks if the key's value exists in the hash-table,
- ❺ the `goto` statement is used to skip aggregation if we encounter an already read observation.

```
437 data _NULL_;
438
439   if _N_ = 1 then
440    do;
441     length curobs 8;
442     drop curobs;
443     declare HASH _obs_(hashexp:16);
444     _obs_.DefineKey("curobs");
445     _obs_.Definedone();
446    end;
447
448   set ❶
449    mysets.INDEXX_OR(
450     where = (date between '01may2015'd ❷
451                       and '30may2015'd)
452    )
453    mysets.INDEXX_OR(
```

```
      where = (country = 'Poland [POL]') ❸
    )
    CUROBS = CUROBS
    end = end
  ;

  if _obs_.check() NE 0 then ❹
    do;
      rc = _obs_.add();
    end;
  else goto SKIPAGGR; ❺

  SoM + measurement;
  i + 1;

  SKIPAGGR:
  if end then
  do;
    put SoM= best32. i=;
    stop;
  end;
run;
```

The SAS log shows similar notes:

```
INFO: Index date selected for WHERE
      clause optimization.
INFO: Index country selected for WHERE
      clause optimization.

SoM=174856439 i=383492
NOTE: There were 98206 observations read
      from the data set MYSETS.INDEXX_OR.
      WHERE (date>='01MAY2015'D
             and
             date<='30MAY2015'D);
NOTE: There were 285681 observations read
      from the data set MYSETS.INDEXX_OR.
      WHERE country='Poland [POL]';
NOTE: DATA statement used (Total process time):
      real time           1.92 seconds
      user cpu time       0.59 seconds
      system cpu time     1.32 seconds
      memory              26405.78k
      OS Memory           41148.00k
```

An additional advantage of that last approach is that it allows us to extend the WHERE clause with multiple ORs in the easiest way by just adding a dataset name with a new WHERE part in the set statement. Which brings the idea of wrapping it into a convenient macro (see the last section for a pointer to details).

BENCHMARKING

The code from the previous section was executed on a laptop with following characteristics:

```
Lenovo Y700,
Intel(R) Core(TM) i7-6700HQ CPU @2.60GHZ,
16GB RAM, SSD + HDD disk drive,
Windows 10 Pro N,
Base SAS 9.4M4 with memsize 8GB.
```

To compare execution times and efficiency the code was also executed on two different data setups and machines: one on a desktop and the other on a server.

Desktop machine characteristics were:

```
HP EliteDesk 800 G1 SFF,
Intel(R) Core(TM) i5-4590 CPU @3.30GHz,
8GB RAM, HDD disk drive,
Windows 7 Enterprise - ServicePack 1,
Base SAS 9.4M4 with memsize 6GB.
```

Datasets were:

```
Small:
 Observations 3'668'464
 File Size (bytes) 353173504 ~ 337MB
Medium:
 Observations 70'304'151
 File Size (bytes) 6765871104 ~ 6GB
Big:
 Observations 378'833'440
 File Size (bytes) 36457152512 ~ 34GB

Common attributes:
 Variables 6
 Indexes 2
 Observation Length 96
```

The results (in terms of time) were as follows:

	Average (Standard Deviation)			
	sql no index	datastep no index	hash appr. 1	hash appr. 2
Small:				
real time	0:00.32 (0:00.02)	0:00.27 (0:00.03)	0:00.30 (0:00.08)	0:00.30 (0:00.03)
user cpu	0:00.11 (0:00.01)	0:00.10 (0:00.03)	0:00.04 (0:00.02)	0:00.05 (0:00.04)
system cpu	0:00.16 (0:00.03)	0:00.17 (0:00.03)	0:00.20 (0:00.00)	0:00.18 (0:00.02)
Medium:				
real time	1:07.23 (0:07.56)	1:27.32 (0:49.28)	0:23.76 (0:01.79)	0:02.55 (0:00.48)
user cpu	0:08.35 (0:00.89)	0:07.32 (0:00.56)	0:00.80 (0:00.07)	0:00.63 (0:00.04)
system cpu	0:07.77 (0:01.08)	0:08.84 (0:02.90)	0:02.53 (0:00.14)	0:01.73 (0:00.16)
Big:				
real time	8:22.38 (0:47.58)	10:09.25 (3:24.73)	1:15.04 (0:09.41)	0:07.33 (0:02.22)
user cpu	0:44.23 (0:03.56)	0:36.22 (0:04.46)	0:03.16 (0:00.23)	0:02.37 (0:00.01)
system cpu	0:42.12 (0:03.54)	1:17.15 (0:10.16)	0:05.71 (0:00.27)	0:03.88 (0:00.29)

Server machine characteristics were:

```
ProLiant DL380 Gen9 HP,
Intel(R) Xeon(R) CPU E5-2667 v3 @3.20GHz,
256GB RAM,
Red Hat Linux,
EG sesion on SAS 9.4M3 with memsize 8GB.
```

Datasets were:

```
Small:
 Observations          4'4019'606
 File Size (bytes) 6709182464 ~ 6GB
Medium:
 Observations          246'134'809
 File Size (bytes) 37513396224 ~ 35GB
Big:
 Observations          1'917'837'577
 File Size (bytes) 292296458240 ~ 272GB

Common attributes:
 Variables             13
 Indexes               2
 Observation Length    152
```

	Average (Standard Deviation)			
	sql no index	datastep no index	hash appr. 1	hash appr. 2
Small:				
real time	0:03.12 (0:00.34)	0:03.52 (0:00.50)	0:03.23 (0:00.75)	0:01.87 (0:00.22)
user cpu time	0:01.79 (0:00.25)	0:02.00 (0:00.32)	0:00.45 (0:00.07)	0:00.48 (0:00.07)
system cpu time	0:01.32 (0:00.09)	0:01.43 (0:00.12)	0:02.45 (0:00.52)	0:01.37 (0:00.16)
Medium:				
real time	0:17.34 (0:01.44)	0:21.41 (0:00.41)	0:12.58 (0:02.35)	0:07.29 (0:00.95)
user cpu time	0:10.04 (0:01.02)	0:12.50 (0:00.18)	0:02.17 (0:00.31)	0:02.22 (0:00.30)
system cpu time	0:07.20 (0:00.41)	0:08.42 (0:00.13)	0:09.24 (0:01.73)	0:04.88 (0:00.60)
Big:				
real time	10:43.32 (0:02.86)	10:22.74 (0:07.88)	0:51.08 (0:14.57)	0:54.05 (0:12.01)
user cpu time	2:15.39 (0:01.03)	2:06.92 (0:05.31)	0:14.21 (0:02.58)	0:15.24 (0:02.16)
system cpu time	1:40.26 (0:02.91)	1:30.86 (0:05.07)	0:35.44 (0:10.96)	0:35.96 (0:08.04)

In both cases index usage in hash-table approaches improved performance time. But to be non-judgemental we have to admit that in the case of "small" sets differences in times weren't as impressive as in the case of "big" ones.

THE CODE

If you are interested in testing approaches presented above yourself and want to play a bit with the code and data you can download SAS codes which were the motivation for this paper under the following "world wild web" address:

`http://www.mini.pw.edu.pl/~bjablons/SASpublic/`

you can find code with data: `Countries.sas` and a bunch of `OR-condition-in-WHERE-clause-with-INDEX-[...].sas` codes (the "`[...]`" extends the discussion).

REFERENCES

[1] Art Carpenter,
"Carpenter's Guide to Innovative SAS Techniques",
SAS Press

[2] Paul M. Dorfman, Koen Vyverman,
"Data Step Hash Objects as Programming Tools",
SUGI 30 Proceedings,
www2.sas.com/proceedings/sugi30/236-30.pdf

[3] Chris Schacherer,
"Introduction to SAS Hash Objects", SAS GF 2015 Proceedings,
support.sas.com/resources/papers/proceedings15/3024-2015.pdf

[4] Paul M. Dorfman, Don Henderson,
"Data Management Solutions Using SAS Hash Table Operations: A Business Intelligence Case Study",
SAS Press

[5] Paul M. Dorfman,
"Fundamentals of the The SAS Hash Object",
SESUG 2016 Proceedings,
analytics.ncsu.edu/sesug/2016/HOW-195_Final_PDF.pdf

[6] Billy Clifford,
"Frequently Asked Questions about SAS Indexes",
SUGI 30 Proceedings,
www2.sas.com/proceedings/sugi30/008-30.pdf

[7] Michael A. Raithel,
"The Complete Guide to SAS Indexes",
SAS Press

[8] Mark Keintz,
"A Faster Index for Sorted SAS Datasets",
SAS GF 2009 Proceedings,
support.sas.com/resources/papers/proceedings09/024-2009.pdf

[9] Paul M. Dorfman,
"The Magnificent DO",
support.sas.com/resources/papers/proceedings13/126-2013.pdf

ACKNOWLEDGMENTS

Author would like to acknowledge Filip Kulon, Krzysztof Socki and Allan Bowe for their contribution and effort to make this paper looks and feel as it should. Thanks Filip! Thanks Krzysztof! Thanks Allan!

Author would like to acknowledge the Citibank Europe PLC Poland for the support in the performance benchmarking execution.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at one of the following e-mail address:

yabwon✉gmail.com

bartosz1.jablonski✉citi.com

or via the following LinkedIn profile:

www.linkedin.com/in/yabwon